ADVMGR Programmer

****
This document contains a bunch of disjointed musings on various subjects pertinent to both programmers and System programmers. The material is based on experience gained in over fifty years of working with computers.

This has been a long and exciting learning journey. It started forty years ago at the old datanex, Inc. working with my friend and fellow self taught hacker/programmer, Paul Klisner. We were busy crashing the company VAX computers nearly every morning, much to the chagrin of the boss. I learned a lot, but times changed, datanex got sold, eventually closed and we moved on.

Over the last twenty or so years, I've managed to get my own alpha (AXP) system and have been porting what I learned over to it. Not always easy. Below is a description of the results.

Along the way, I've met and often made friends with some very smart and very nice folks. It seems that OpenVMS has not gone away and may well be making a resurgence. Well it should, it's often the best OS available for many business and engineering/scientific operations.

This is an attempt to share some of my hard won knowledge. I make reference to many routines called from system programs. These are found in MARD:MARLIB.MAR which has been included in the ADVMGR distribution. Please refer to it for details. Other sections refer to software not distributed with ADVMGR. It can be supplied on request.

##########
Secondary page/swap files
##########

By default, the system creates and installs a page file, a swap file, a dump file and an error log file on the system disk. The size of these files depends on the amount of memory the system has plus a few other factors and can be rather large. The amount of room they take up increases dramatically in a cluster situation where multiple nodes are booted from the same system disk, the normal mode of operation.

This means that the system disk is guaranteed to be active in a normal stand alone system and very active in a cluster. This affects the performance of the system noticeably. It can be desirable to make use of one or more so called secondary page and swap files located on a different disk. Ideally the choice would be to have a disk for each node which is high speed, attached only to that node and mostly dedicated to those two files.

Below are some code and commands for performing this task. A few comments are appropriate. First, the commands in SYS$SYSTEM:MODPARAMS.DAT are root specific. The code for SYS$MANAGER:SYPAGSWPFILES.COM may not be. On my system, SYPAGSWP was located in dev:[SYSn.SYSCOMMON.SYSMGR] which means it was common to all nodes. It was necessary to delete it from that area and recreate it in dev:[SYSn.SYSMGR] for each node. SYPAGSWPFILES.COM is run very early on in the boot process and, once correct, is not changed. MODPARAMS.DAT needs more explanation.

```
!PAGEFILE1_SIZE=0
!SWAPFILE1_SIZE=0
!PAGEFILE2_NAME="$2$DKA200:[PS]PAGEFILE_TULASI.SYS"
!PAGEFILE2_SIZE=3000000
```

```
!SWAPFILE2_NAME="$2$DKA200:[PS]SWAPFILE_TULASI.SYS"
!SWAPFILE2_SIZE=50000
PAGEFILE=0
SWAPFILE=0
```

Setting the primary page and swap files to zero size is tempting but a bit dangerous. If the secondary disk cannot be mounted for some reason, you can't boot and fix things. Better to let AUTOGEN produce a primary file large enough to boot the system but small enough to save much space and make good use of the secondarys.

The commands for name and size should be executed once to create the files. Never run AUTOGEN with the parameter REBOOT. Do the following:

```
$ @SYS$UPDATE:AUTOGEN SAVPARAMS SETPARAMS FEEDBACK
$ TYPE/PAGE SYS$SYSTEM:AGEN$PARAMS.REPORT
```

Read the report. It gives you lots of information including errors you can correct before you produce an unbootable system. Set the size of the secondary files to be more than adequate for a single page or swap file.

Once created by autogen and before you reboot the system, comment out your name/size commands and uncomment the PAGEFILE=0 commands. These will stop future runs of AUTOGEN from trying to create new files.

SYPAGSWPFILES.COM should contain the following code:

```
$ LOOP1:
$ ON WARNING THEN GOTO LOOP1
$ WAIT 0000 00:00:00.50
$ READY = F$GETDVI("$2$DKA200:","AVL")
$ IF READY .EQS. "FALSE" THEN GOTO LOOP1
$ SET NOON
$ MOUNT/SYSTEM/NOASSIST $2$DKA200: EXTRA
$ MCR SYSGEN
INSTALL $2$DKA200:[PS]PAGEFILE_TULASI.SYS/PAGEFILE
INSTALL $2$DKA200:[PS]SWAPFILE_TULASI.SYS/SWAPFILE
EXIT
```

AUTOGEN is smart enough to see that the secondary file is more than adequate for paging (swapping) and will calculate and resize the primary file on the system disk to a minimum necessary for booting. The OS is smart enough to use the secondary file to the maximum. On a busy system, the performance gain can be significant.

The CRASHDUMP and ERRORDUMP files are big enough that you can save a good bit of space on the system disk by relocating them to another disk. The process is a bit complicated, on Alpha and Integrity systems requires variable definitions on the console that are not very obvious and subject to error and, depending on the particular machine/emulator may not work at all. While it can save some space on the system disk, it will not effect performance at all. On my system I kept them on the system disk.

```
################################################################################
Search lists.
################################################################################
```

For my own system, it has proven advantageous to define some logicals as a search list. I define them in the JOB_TABLE. The total number of logicals you may create

is dependent first on JTQUOTA variable in the UAF file which, should you encounter the “exceeded quota” error or stuff is not defined that should be, can be doubled from 4096 to 8192 or even more. Secondly, the SYSGEN parameter PQL_DJTQUOTA can be modified in MODPARAMS.DAT followed by running AUTOGEN. I multiplied the default by ten.

When defining/creating search list logicals, remember that each equivalence string is limited to 255 characters and there can be no more than 128 such strings. Actually trying to make a list like that would run you out of quota space in a very big hurry and out of system memory shortly thereafter. Keep them short. LOGIN.EXE limits the number of definitions per logical to 20. Line continuation is possible for this table only. You may define several search logicals and then combine them by making the final logical name a list of the logical names you have just defined. Please read the directions and see the examples in ADVMGR:LOGIN.DAT.

##
Executlets.
##

Please note: All work involving executlets is based on code written by Brian Schenkenberger (aka VAXman) of Tmesis. There is more work described later on that required my contacting him. He was most cooperative, gracious and generous with his time. Many thanks.

IMPORTANT NOTE!!!! The addressing page size or boundary on the VAX is always 512 bytes. On the Alpha, it is larger and varies with the machine. The Alpha page size, F$GETSYI(“PAGE_SIZE”) ranges from 2^13 (8K) to 2^16 (64K). It appears that the page size for the Integrity series is always 64K. Brian has written a cute one liner piece of DCL found in my LOGON.COM file and the KITINSTAL.COM file distributed with the Advanced System Manager Console software that determines the value used in linking special software.

The software that involves Executlets including the code that sets the prompt to the default directory, that is, code loaded into the operating system, benefits from using the actual value for the machine on which it’s run. A value of 64K works on all machines but, on smaller machines wastes a great deal of addressing space. Therefore, when we install ADVMGR software on your computer, we determine the correct value for /BPAGE and use it. This value ranges from 13 to 16 inclusive and could be different if you upgrade your alpha system. Re-installation of ADVMGR and your software might be necessary.

AST_LOAD.EXE and AST_UNLOAD.EXE are used to load and unload special code to/from system memory. It is the compiled and linked "executlet" file that serves as a substitute for the old VAX "copy the code to S0 and queue an AST to it" that does not work on ALPHA. The "executlet" code, linker switch and option files can be supplied on request.

You can add your own executlet code to this file. It will hold as much as you want and contains a good deal of reusable (common) code for the piggy_back return ASTs. I've made it simple enough that, with a little patient reading, it should be clear in its operation. When stable, it should be loaded at boot. Note! It is done this way to allow the system to queue an AST. Putting the code into an "executlet" means there is no address relocation required as there is no linkage psect. The loading system subroutine takes care of addresses. No linkage means there is another mechanism for queuing the AST. It is the same as done for the OS executlets (system API's) that are loaded during the system boot process.

The loader adds in special code that handles queuing the AST and, when you call the code at the address returned from FIND_LDRIMG in the shared library MARLIB,

you are actually getting the address of this special code. This code queues an AST to your victim and eventually, in the process of changing mode to kernel, gets your victim to run your code. (Whew! That was a mouthful). It actually makes writing the code easier.

Making use of execulets is demonstrated in DEFDIR, GETCMD, DISMOUNT, FORCEPRV and PURGWS. The source for DISMOUNT is included in the installation of ADVMGR.

##
Multi-sessions.
##

BOSS.EXE is 'multi-session on a single terminal' code. It should be installed with the non-threatening privileges of PHY_IO and SHARE so everyone can use it. It has an advantage of being able to buffer some output so that the screen can be refreshed on switching processes.

Users who will be in menu driven software (typical business) and have a need to use two terminals/windows, will be well served with BOSS properly started at login. An example might be a user who is at the counter filling out an invoice for a customer and has a need to look up an inventory item to see if it's available. Use the following setup in LOGON.COM:

```
$ BSL[0,8] == 28
$ CI[0,8] == 9
$ CJ[0,8] == 10
$ S1 == "'''BSL'''CI'''BSL'''CJ'YOUR MAIN MENU"
$ S2 == "'''BSL'''CI'''BSL'''CJ'YOUR MAIN MENU"
$ BOSS/START=(A,B)/STUFF=("'''S1'","'''S2'")
$ EXIT
```

This creates two sub-processes, each running the main menu. The user switches between them by typing CTRL-\ followed by A or B. The screen will be refreshed automatically with this setup.

The disadvantages are that BOSS is full screen even on a terminal emulator and logging out is a bit complicated. You also have to remember which screen, A or B, you are on.

SWIM.EXE is also ‘multi-session on a single terminal’ code. It should be installed with the non-threatening privileges of PHY_IO and SHARE so everyone can use it. It is pleasant to use in that there are multiple "windows" on the terminal you can switch between/among with CTRL-]. The windows can be moved and resized. They will overlap and will repaint when selected.

If you are running business applications using terminal emulation (Putty?) software which can present a screen much larger than the VT100 standard 24 X 80, you may prefer, as we do, using SWIM with its 24 X 80 terminal windows. Each window can be maximized to fill the entire emulated terminal window using CTRL-\ X and quickly returned to their previous size and positions using the same two key sequence. Use the following setup in your LOGON.COM:

```
$ WINDOWST
$ IF FT
$ THEN
$   do some stuff and run a program
$ ELSE
$   WINDOWS
$ ENDIF
```

```
$ SET TERMINAL/APP
$ EXIT
```

The above is good for users who are menu driven. In my own case:

```
$ WINDOWST
$ ...
$ IF .NOT.FT THEN $ SEDIT     ! Start my editor as a sub-process.
$ ...
$ TSR
```

The symbols WINDOWST, FT and WINDOWS are defined in SYS$MANAGER:SYLOGIN.COM. They execute scripts in my area that test for the FTAx: terminal driver, set the global symbol FT as 0 or 1 and do other things that make life easier. In the most simple form, modify your LOGON.COM file to do WINDOWST early on and then test FT and don't start the editor as a sub-process if true. It saves on process slots and avoids unnecessary clutter. If I want a couple of windows, I simply type WINDOWS at the command prompt.

Of course there is another, very simple way for obtaining multiple sessions. Just start a new Putty session and log in again.

##
LOGON.COM
##

Defining symbols, logicals and keys in your LOGON.COM file can get carried away. Years ago, it would take me twenty seconds or so to log in and logging in placed a noticeable load on the system. Each definition had to endure the overhead of image startup and rundown; significant on VMS. Hunter Goatly wrote a program that did all this in one place. This program was my introduction to system programming and I will be forever grateful to Hunter for his contribution. Recently I added his key definitions and a choice of putting logicals in the PROCESS or JOB table as well as the ability to make logicals with multiple equivalences (search logicals). These changes made it even more useful, but it could only work for users with the CMEXEC privilege. This excludes a large number of users.

To make it more general, I rewrote LOGIN.MAR which, rather than compiling the users definitions, interprets them. Thus LOGIN.DAT can be place in the users default login directory, modified by him as he pleases and LOGIN.EXE can be installed with the CMEXEC privilege and run by any user. It will greatly reduce stress on the system as well as stress on the user.

##
SYSTEMS PROGRAMMERS!
##

If you are a systems programmer, you know that you need to be very careful not to try and manipulate a device/process that isn't there. It crashes the system. You must also lock appropriate databases and raise IPL, etc. checking for errors and handling them in an appropriate fashion. It is easy to overlook a check that you should have made or to program in such a way that you either can't do what you want or you crash the system.

We have produced four (five?) routines that help this process a great deal in that they are tested tried and, we believe, true. They are:

GRAB_TERMINAL
GRAB_DEVICE

GRAB_PROCESS
GRAB_KAST

And another subroutine to be used with the first, TH_STUFF. They are all in MARLIB, the shared library (DLL). The naming convention for the fourth is misleading at best but I wanted to group them as they are used in a similar fashion. For each of the first three, you do:

```
      $CMKRNL_S -
            ROUTIN = GRAB_XXX, -
            ARGLST = KARGS
```

KARGS varies with the situation and is described in detail in MARLIB and example programs. The first three are similar. They attempt to gain exclusive access to a valid named device/process, performing all the desired checks along the way. If successful, they will then execute a named JSB subroutine in your program whose address you passed in the argument list. They will then release everything appropriately. By the way, it is up to you to make sure everything is locked in memory so there are no page faults. We're operating at high IPL. Do this by using my macros, ILOCK/IUNLOCK.

Should the grabbed device be a terminal of TT class driver type, and the name of the JSB routine to be executed is TT_STUFF (place a string into the type-ahead buffer of the terminal), the JSB routine is in MARLIB and will use a passed address of the string descriptor you desire.

The difference between GRAB_TERMINAL and GRAB_DEVICE is that GRAB_TERMINAL includes checks that apply only to terminals. Use the wrong one and it's crash city.

GRAB_KAST is slightly different in that it is not getting control of another device/process but rather building an ACB in S0 space so an AST you've written and put in AST_LEI.MAR will be executed by the intended process. There is an option to execute a JSB routine in your program, should you need to put some information into the ACB prior to your victim executing the AST. See DISMOUNT.MAR as an example.

Another difference is that you will use two MACRO's, KAST_OFFSETS and KAST_ARGUMENTS in your program. The optional subroutine is provided in a unique place and used rather differently from the other GRAB_XXXX routines.

These routines are used in most of my systems programs and make for a dependable execution environment and much easier programming. For more details, examine my source code for routines in the menu. Note! If you add your routine to AST_LEI.MAR, please remember to update KAST_OFFSETS in MACROLIB.LIB and rebuild the .MLB's.

##
Symbionts.
##

A symbiont is an executable program that attaches itself to a queue through the queue manager and which processes in some manner, that which has been placed in the queue. Two examples supplied by VMS are the batch queues and the print queues. The batch queue symbiont simply creates a new detached process and then uses the contents of the queued file as a DCL command source. This constitutes a “Batch Job”.

The print symbiont usually does more in the way of processing what is in the queued file on its way to a printer device.

Sometimes there are advantages to being able to queue files to a named queue for further and possibly sequential processing. Rather than running some batch process passing it parameters that will eventually be processed by the batch queues or doing the processing by hand, A simple “$ COMMAND FILE” is often the most simple.

John Osudar of Argonne National Laboratory wrote a symbiont that is intended for general processing. It is well suited to this in that it utilizes a user supplied DCL procedure to do the processing. This procedure can do pretty much what it pleases.

In our environment, printing is difficult. What we have done is to write a Windows program that periodically scans the default folder for the Windows FTP server process for files with the .LIS extension. On the VMS system, we use the “$ PRINT FILE” command to queue up the file to be transferred to Windows for printing. The SYS$PRINT queue is initialized with the processor being the EXECSYMB symbiont and our DCL processing script, REMPRT.COM, as a parameter. RMTPRT.COM copies the submitted file to one with the .LIS extension before FTPing it to Windows.

We have extended this by setting up two additional queues for PUTing and GETing files to/from the Windows FTP default directory. We define FPUTA, FPUTB, FGETA and FGETB for sending/fetching both ASCII and binary files. It is convenient. FGETx is more complex. Using the QUEUE will not work unless the file to be fetched already exists in the default directory at the time the command is issued (the nature of SUBMIT.EXE on which it is based). To get around this, FGETA points to a DCL script of the same name that either uses the QUEUE or does the FTP operation directly in the case that either the file is not present or you use a wild card.

At the end of R$:[COM]OUR_SITE.COM, which is called from SYSTARTUP_VMS.COM on startup, is a line that points to the script, @R$:[COM]QOPERATIONS.COM. You may modify the script to suit your needs.

In our case we make sure that this script is running on the primary node of the cluster or a stand alone node by checking that SYS$TOPSYS is SYS0. The remainder of the script is very specific to our needs but gives some good examples of using queues and EXECSYMB.

##
TCP/IP NTP Time Service:
##

I discovered that Windows does not like you to use numerical IP addresses to reach the internet when you are doing it from AXP emulators, at least some of them. As soon as I properly configured the BIND in TCP/IP core services with a host name in the Hosts file, everything seemed to work.

If you are in a cluster, it is probably best to use one node as the node that reaches out over the internet to get the time and have it broadcast to the other nodes. In the primary configuration file, you need at least one “server ‘Time Service URL’” line and then one “broadcast x.x.x.255” line. In my case the x’s are 192.168.1.255, the first three numbers representing my Local Area Network.

On the other nodes, you should have a server line that points directly to the primary. In my case: “server tulasi”. Start the time service on all nodes and things go well. You will find that the time on all the nodes is very accurate and even more precise except under the most adverse of circumstances. Make sure your

HOSTS file has your local addresses and names set up correctly.

Daylight savings time changes seems to confuse the time service. Politicians keep messing with daylight savings and I don't trust the automatic VMS procedure. Just setting the new time by hand yields a "sanity check" in NTP. This will continue on restart. The best way, in my opinion is to use the DCL procedure;

@SYS$EXAMPLES:DAYLIGHT_SAVINGS.COM

It will adjust the system time cluster wide plus or minus one hour when run.

##
P2_COMMON User Space:
##

P2_COMMON space needs a bit of discussion. I have an MARLIB (dll) callable subroutine that makes use of it and is fairly flexible.

On AXP boxes (and Integrity I assume) there is a very large address space not used by VMS. It simply does not exist on the old VAX. The 32 bit OS uses addresses ^x00000000 through ^x7fffffff for user space and ^x80000000 through ^xffffffff for the operating system. In operating system space, addresses would be negative in two's compliment arithmetic. User space is divided into two areas, P0 - ^x00000000 through ^x3fffffff for programs and P1 - ^x40000000 through ^x7fffffff for control, DCL, symbols, logicals and stack space.

The AXP box is 64 bit and puts the operating system in negative address space just like on the VAX. It does so by extending the high order bit of the lower 32 bit address into the upper 32 bits. This leaves addresses ^x00000000 80000000 through ^x7fffffff ffffffff as unused and inaccessible to a 32 bit OS. This is a lot of space, about 10^18 bytes.

The later versions of VMS provide system services for accessing this space. SYS$CRMPSC_PFN_64 is one example and the system service call I use in the MARLIB subroutine called P2_COMMON. My code can allocate about two million 8192 byte CPU pages. The addressing is tricky and requires a passing familiarity with MACRO64 assembler language. This is why I provided the routine. It's relatively easy to use and performs a variety of useful operations for you. One of the benefits is that you can allocate quite a chunk of memory for the personal use of your process and you will not have to worry about running out of program space or worry about having your stack overwrite your data or your data overwriting symbol and logical space.

P2 space, like P1 and P0 space, is process permanent. The addresses you get using SYS$CRMPSC_PFN_64 disappear at image rundown. However, the contents are not zeroed when space is commandeered, so you may pass information from one program to the next, if you are careful to use identical parameters in the call.

##
Decwindows:
##

WML = Windows/Mac/(31 Flavors of Linux)

I am very aware that the modern computing paradigm insists on having windows and a mouse. Ever since Gates and Jobs stole the idea from Xerox, people have been convinced this was the new way of doing business. You could have two or more programs running simultaneously and switch between them with a click of the mouse. This is a very handy thing to have. The mouse made highlighting text simple. A lot of effort and time have been spent on rather elaborate word processing programs,

spread sheets and other programs for producing elaborate output for consumer consumption.

There are things people need to do in a very large area of business and personal life for which WML computers are indispensable. They are not going away.

That does not mean they are ideal for all areas of business. Having two or three windows running separate programs open at the same time is very desirable for a lot of business operations. It is easily accomplished in VMS without elaborate windows.

People are accustomed to using a mouse to the point that they do not realize how much time is wasted using the mouse to select text boxes for the next item of input. It is not efficient. In WML programs, the screen layouts are often confusing and/or misleading while the old numbered input with masked data entry items are very easy to understand and moving from item to item with the tab key is very efficient, much more so than using a mouse.

Individuals are not going to spend the money on OpenVMS, nor should they. Business and engineering operations of some size will benefit greatly from the fact that OpenVMS is so efficient and safe when properly configured. Sometimes the old fashioned way is better. It depends on the circumstances.

I have Decwindows running on my system. I find it basically useless. I will not adapt my software to a useless, high overhead paradigm that simply gets in the way.

##
Data Entry Routines
##

The input routine is especially designed to behave in a way similar to old fashioned heads down data entry on an IBM terminal. You can declare the type of input accepted and the minimum and maximum length. Errors are flagged immediate for correction, there is no autotab (extra characters are simply not echoed and ignored and escape sequences are swallowed whole and ignored. The tab character is interpreted as "close the field and move to the next field". There are only two edit operations: 1) CTRL U which means start over and the backspace key to erase the characters to the left. The paint character is "." and the input and output can be masked, e.g. "(...) ...-...." for phone number.

Post processing such as compression and parsing with capitalization with appropriate re-display is available. Numeric input can be re-displayed right justified, right justified with a period and right justified with period (dollars) and commas.

These routines can be called from any language, they are written in MACRO32 and are very efficient. They allow you to do things that higher level languages either cannot do or do so extremely inefficiently.

##
THE MACRO LIBRARY, MACROLIB.LIB & MACROLIB64.LIB
##

Macros for VMS MACRO32 & MACRO64

The VMS assembly language is a powerful set of complex instructions which, when combined with the library of system services (executive images), allow the competent programmer to write code that can do just about anything for which the author has privilege. Although a dying art, assembly language programming has its place for times when the maximum efficiency, speed and compactness is

needed. In these cases, the skill of the programmer and his knowledge of how the assembly instructions work, can produce results about which the high level language programmer can only dream.

I remember back at datanex, Inc. our company was writing EZBRIDGE for transferring files between a VAX and an AS400. This required translation of ANSII to EBCDIC and back. They wrote it in "C". Translation took forever (seconds) and really slowed the file transfer process noticeably. I wrote a routine in MACRO32 they could call which required one instruction to do the translation for one character. They were amazed.

A programmer will likely choose assembly language for so called systems programming wherein certain abilities not provided by the operating system and its executive images are needed. These cases, which are by far the most common out in the real world, do not really require compactness and speed. Developed system programs usually are used infrequently. What is really needed is maintainability which means readability. The macros described below provide this.

Now I've heard all the arguments about how "unnecessary" all of this is and how it just adds more "complexity" to the process. Back in the days before VAX, we were writing business programs in DBL from then named DISC (now SYNERGEX) and had the usual spaghetti code and associated headaches. I wrote a pre-compiler program that used exactly the structured approach described below. Ken Lidster hated it but eventually introduced CASE and IF/THEN/ELSE structure to DBL.
My structure keywords give the reader a sense of flow that is easy to understand because it is exactly how people think. My own experience was that, as soon as my team got over the initial shock of something new, they loved it. Code became readable. Bugs were dramatically reduced. Bugs were more easily found and productivity quadrupled in a matter of a couple of months. So before you dismiss my approach out of hand, read some of my code.

*****

We have designed these macros so that they appear to look alike and function the same for both MACRO32 and MACRO64 The differences are:

1) MACROLIB64 uses three scratch registers whose contents are unknown to the programmer after the invocation. They are: R0, R1 and R23. MACROLIB32 does not alter registers.

2) In MACROLIB64, branches can take you anywhere so the "W" (word branch) form available in MACROLIB32 isn't used.

The macros provided are designed to turn assembly language into a structured language in that there are no visible branch/jump instructions in the program and, except for the entry name, no labels are required. A program might have the rough appearance of:

```
NAME::
.CALL_ENTRY...
INSTRUCTION(S)
REPEAT
        INSTRUCTION(S)
        UNTIL   condition
        INSTRUCTION(S)
        IF      condition
                INSTRUCTION(S)
        ORIF    condition
```

```
                INSTRUCTION(S)
        ORIF    condition
                INSTRUCTION(S)
        ELSE
                INSTRUCTION(S)
        ENDIF
        INSTRUCTION(S)
        WHILE   condition
        INSTRUCTION(S)
ENDR
INSTRUCTION(S)
RET

.END NAME
```

Repeats can be nested indefinitely and if/orif/else/endif to twenty-five.
Actually, any depth greater than about five renders the code unreadable anyway.

The following is a list of the macros and their arguments:

REPEAT
    This forms the beginning of a loop. It takes
    no arguments and does not produce any code.

UNTIL ARG1, condition, ARG2, LENGTH
    Until the condition is met, repeat the loop (go back to the beginning).

        ARG1 and ARG2 are any valid location
        in any valid addressing mode.
        Condition is the comparitor. It can be:

                EQL
                NEQ
                GTR
                GEQ
                LEQ
                LSS

        Each of these may have a 'U' appended for testing
      unsigned. It would be meaningless for EQ & NE.

        LENGTH is the address size. For example:
        UNTIL @(R0), GTR, R10, W
        will compare words. Options are:
      B, W, L and Q

WHILE   ARG1, condition, ARG2, LENGTH
        While the condition is met, continue execution.
        Otherwise break out of the loop and resume
        execution after the ENDR.

BREAK
        Unconditionally breaks out of the loop. Handy
        when you do an IF and some stuff and then
        terminate the loop.

CONTINUE
        Unconditionally branches to the beginning of the loop.

ENDR
The end of the loop. If execution reaches here, it branches back to the REPEAT of the same level of nesting.

IF ARG1, condition, ARG2, LENGTH
If condition is true, execute the code between the IF and the next ORIF/ELSE/ENDIF.

ORIF ARG1, condition, ARG2, LENGTH
If condition is true, execute the code between the ORIF and the next ORIF/ELSE/ENDIF. Multiple ORIF's are possible between an IF and ENDIF.

ELSE
Execute the code between the ELSE and the next ENDIF. An ORIF after an ELSE makes no sense and will never be reached. It also won't compile.

ENDIF
Terminates this level of the IF sequence. Required for each level of IF.

Discussion:

For the sake of efficiency, if the ARG2 value is identical to #0, a TST instruction is used rather than the CMP for MACRO32. For MACROLIB64, it is a simple branch on register test.

There are versions of the macros where no CMP or TST instruction is produced but rather a branch is created that assumes an instruction has just been executed that sets the condition codes. The macros are:

MACROLIB32:
```
IFC     condition
ORIFC   condition
UNTILC  condition
WHILEC  condition
```

MACROLIB64:
```
IFC     condition,register
ORIFC   condition,register
UNTILC  condition,register
WHILEC  condition,register
```

In other words, those macros that have ARG1 and ARG2 arguments have this form as well.

The following macros may have the letter W appended to the names in order to produce a word branch rather than the default byte branch. Compile errors that indicate branch out of range may be solved by appending the W: This is not part of MACROLIB64.

```
UNTILW
UNTILCW
WHILEW
WHILECW
BREAKW
CONTINUEW
ENDRW
```

```
        IFW
        IFCW
        ORIFW
        ORIFCW
        ELSEW
```

Additionally, we have IF_ERR and IF_OK followed by ENDIF for checking the low order bit of R0 after a call or test.

These macros, while very efficient, cannot handle all cases where there is a strong need for compactness and/or speed. But they more than make up for this in readability and maintainability.

Other macros of interest:

The following macros call the library routines which lock and unlock the entire image. On the AXP system, this is usually the most safe and efficient way of locking code in memory because you have to include the linkage psect(s) which can be difficult to do.

```
ILOCK       locks the image and prevents page faults.
IUNLOCK     unlocks the image.
```

Error checking with ERCK [string],[destination],[branchword]

Error checking usually looks something like:

```
      IF_ERR
            JSB   DSPERR
            RET
      ENDIF
```

or something to that effect. We have a more generalized macro that takes arguments and does some (but not all) operations needed. The macro always produces the "IF_ERR" and ENDIF" pair. The first argument is a string of characters (upper case) that can contain the following letters in any order:

[D],B,R,S

B, R and S are mutually exclusive in that B produces a BRB to the second argument. R produces a RET and S produces a RSB. The D does a JSR DSPERR in MARLIB followed by MOVZBL #SS$_NORMAL, R0. If a third argument is provided and a B is used, the branch will be a word branch.

If no arguments are given, a RET is produced.

****

RMS (Record Management System)

RMS is one of the best pieces of software DEC wrote. It is awesome. It is jaw droppingly versatile and supports every type, style and variation of file you can imagine and a few you can’t.

It is also horrifyingly complex. While the documentation defines the variables, it does not explain much about their use and which ones go together to produce a given result. Examples are both sparse and esoteric, leaving the user to either drop back to a high level language or find his way alone. It took me days of reading, searching and just plane trial and error to figure out how to read and write a simple variable length text file in MACRO32.

```
OPENR       name, address of file spec descriptor, [address of default file
descriptor]
REOPENR     name
OPENW       name, address of file spec descriptor, [address of default file
descriptor]
REOPENW     name
OPENA       name, address of file spec descriptor, [address of default file
descriptor]
OPENI name, address of file spec descriptor, address of record buffer
      descriptor, [address of default file descriptor],
      [<OR'ed additional operation bits enclosed in angle brackets>]
READ        name, address of record buffer descriptor
WRITE       name, address of record buffer descriptor
CLOSE       name
ERASE       name
```

That is exactly what the above macros do. They take care of all structures ($FAB, $RAB) and put them in the PF$DATA psect, same as the PRINTF macro described below. The OPENI is for ISAM files and the record operations variable are the standard: RLK ASY and SYNCSTS bits needed to update with record locking. The OPERATION argument allows the user to add bits for special purposes (see LOCKHOLDER.MAR).

When you reopen a file, please use exactly the same name or things will get confused.

If special operations are required or information needed, the naming convention for the $FAB and $RAB is: the name you give as the first argument of the macro with an "F" appended for the $FAB and an "R" appended for the $RAB.

*****

The macro PERR takes one argument, a single character and calls SERR. R0 is used and preserved. It writes to the system error log and works in kernel mode.

The following macros make it a bit easier to use the LIB$SYS_FAO library call or the call to DECC$GPRNTF in the MACRO64 world.

```
PRINT <delimited character string>
PRINT <delimited character string>,\MN    ; MACRO64
```

Stores the character string in a psect called PF$DATA. Can be used anywhere in a CODE PSECT because it saves and restores the current PSECT.

```
PRINTF      <delimited character string>,arg1,arg2.... or
PRINTF      <delimited character string>,\MN,arg1,arg2.... ; MACRO64
```

Stores the character string in a psect called PF$DATA. Can be used anywhere in a CODE PSECT because it saves and restores the current PSECT. In MACRO32, the arguments replace FAO arguments embedded in the string (see below for some FAO arguments).

For MACRO64, the character string uses "C" formatting.

```
!AC     Inserts a counted ASCII string. It requires one parameter: the
        address of the string to be inserted. The first byte of the string
        must contain the length (in characters) of the string.
```

!AD Inserts an ASCII string. It requires two parameters: the length of the string and the address of the string. Each of these parameters is a separate argument.

!AF Inserts an ASCII string and replaces all non printable ASCII codes with periods (.). It requires two parameters: the length of the string and the address of the string. Each of these parameters is a separate argument.

!AS Inserts an ASCID string. It requires one parameter: the address of a character string descriptor pointing to the string. $FAO assumes that the descriptor is a CLASS_S (static) or CLASS_D (dynamic) string descriptor. Other descriptor types might give incorrect results.

!AZ Inserts a zero-terminated (ASCIZ) string. It requires one parameter: the address of a zero-terminated string.

!XL Converts a long word value to the ASCII representation of the value's hexadecimal equivalent. It requires one parameter: the value to be converted.

!ZL Converts an unsigned long word value to the ASCII representation of the value's decimal equivalent. It requires one parameter: the value to be converted.

There are a lot more. See the Systems Services Reference Manual under $FAO.

In addition, for MACRO64, we have PUSHR and POPR macros which take an identically ordered string of comma separated registers to be pushed and later popped onto the stack. This is an easy method of saving a few registers temporarily. It is slightly more efficient to "roll your own".

PR_C This macro takes the argument of one of the colors as defined in SMAC and causes the escape sequence of that color to be placed where R10 points. It uses the subroutine INSERT_COLOR in MARLIB.

##
Final comments.
##

Please note that MARLIB.EXE is installed /SHARE/PROT. It has several entry points which are called by changing mode to kernel and must be protected from being written from user or executive mode. And, by the way, I discovered that you can install an executable image with privileges but not a shareable image.

This is the end of my musings of this type. If you want some thoughts on security, please read my web site.

All of this work performed over decades, is a labor of love. It is yours for free. I can provide many example programs on request and am available for consulting. As I have stated earlier, I think VMS is the best and most safe general purpose time sharing operating system in existence.

Rick Marsh Ph.D.
Tulasi Software Systems
www.rickmarsh.com
rick@rickmarsh.com